Point Reprojection and Dynamic Scenes

Erik Reinhard[1]

April 24, 2001

[1]University of Utah

Contents

1 Introduction

Given the feasibility of interactive ray tracing, the following chapters deal with algorithms that allow higher scene complexity and/or larger image sizes, as well as algorithms that enable interactive manipulation of scene geometry.

In the following chapter, a high-level optimization is presented for increasing image resolution and scene complexity. This approach implements techniques that allow results from previous frames to be reused. There are a number of techniques in existence that can be employed. Reusing previous results involves displaying pixels at a higher frame-rate than new samples can be produced. Storing pixels in a 3D point cloud and reprojecting the points for each new frame is one such method. Several choices need to be made to optimize such reprojection techniques, including strategies to decide which pixels to render for the next frame, whether frame-based rendering is going to be used in the first place or whether the algorithm is to operate in asynchronous mode (frameless rendering). Also, the point reprojection may operate in parallel or serially by a single front-end processor. Chapter 2 presents an implementation and discusses its merits and weaknesses.

A second wish users may have once they've implemented an interactive ray tracer, is extending interactive ray tracing to allow dynamic scenes. Walking through a scene at interactive rates is useful, but being able to interact with it is even more useful. While ray tracing is traditionally good at rendering static scenes, we show that it is possible to render animated scenes, or even interact with the scene in real-time. This requires some basic modifications to the ray tracing algorithm (in particular the spatial subdivisions need a tweak) that are easy to implement, incur a small performance penalty, but allow animation to take place interactively and give the user the ability to manipulate objects. Chapter 3 shows the details, while Appendix A contains an original paper on this subject.

2 Point reprojection

Interactive Whitted-style ray tracing has recently become feasible on high-end parallel machines [14, 17]. However, such systems only maintain interactivity for relatively simple scenes or small image sizes, due to the brute-force nature of these approaches. While keeping the algorithm as simple as possible is an important factor for their succes, reasonably straightforward extensions have been deviced to improve visual appearance for much larger image sizes and scene complexities. After a brief overview, one such system is explored further in this chapter.

By reusing samples instead of relying on brute force approaches, the limitations in scene complexity and image size can be overcome. There are several ways to reuse samples. All of them require interpolating between existing samples as the key part of the process. First, rays can be stored along with the color seen along them. The color of new rays can be interpolated from existing rays [3, 12]. Alternatively, the points in 3D where rays strike surfaces can be stored and then woven together as displayable surfaces [21]. This method was designed to display course results by a display processor while new samples are created by a rendering back-end which can consist of one or more renderers. As new results become available to the display processor, the image is refined and redisplayed. Finally, stored points can be directly projected to the screen, and holes can be filled in using image processing heuristics [25]. All techiques that re-use samples rely on the fact that the reprojection step is much cheaper than the generation of new samples and are therefore typically employed in cases where sample generation is too slow for creating interactive results. In the case of Simmons' work, this occurred because the lighting simulation is too complex for interactive display [21]. Walter's point reprojection algorithm is directed towards interactive display of scenes that are too complex to display interactively otherwise.

Another method to increase the interactivity of ray tracing is *frameless rendering* [4, 6, 17, 27]. Here, a master processor farms out single pixel tasks to be traced by the slave processors. The order in which pixels are selected is random or quasi-random. Whenever a renderer finishes tracing its pixel, it is displayed directly. As pixel updates are independent of their display, there is no concept of frames. During camera movements, the display will deteriorate somewhat, which is visually preferable to slow frame-rates in frame-based rendering approaches. It can therefore handle scenes of higher complexity than brute force ray tracing, although no samples are reused.

The main thrust of this chapter is the use of parallelism to increase data reuse and thereby increase allowable scene complexity and image size without affecting perceived update rates. The remainder of this chapter uses the *render cache* of Walter

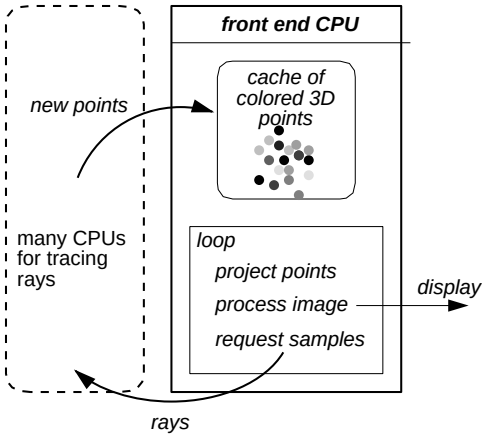


Figure 2.1: *The serial render cache algorithm [25].*

et al. [25] and applies to it the concept of frameless rendering. By distributing this algorithm over many processors we are able to overcome the key bottleneck in the original render cache work. We demonstrate our system on a variety of scenes and image sizes that have been out of reach for previous systems. The work described in this chapter is currently under submission for the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics [19].

2.1 Render cache algorithm

The basic idea of the render cache is to save samples in a 3D point cloud, and reproject them when viewing parameters change [25]. New samples are requested all over the screen, with most samples concentrated near depth discontinuities. As new samples are added old samples are eliminated from the point cloud.

The basic process is illustrated in Figure 2.1. The front-end CPU handles all tasks other than tracing rays. Its key data structure is the cache of colored 3D points. The front end continuously loops, first projecting all points in the cache into screen space. This will produce an image with many holes, and the image is processed to fill these holes in. This filling-in process uses sample depths and heuristics to make the processed image look reasonable. The processed image is then displayed on the screen. Finally, the image is examined to find "good" rays to request to improve future images. These new rays are traced by the many CPUs in the "rendering farm". The current frame is completed after the front end receives the results and inserts them into the point cloud.

From a parallel processing point of view, the render cache has the disadvantage of a single expensive display process that needs to feed a number of renderers with sample requests and is also responsible for point reprojection. The display process needs to insert new results into the point cloud, which means that the more renderers are used, the heavier the workload of the display process. Hence, the display process quickly

becomes a bottleneck. In addition, the number of points in the point cloud is linear in image size, which means that the reprojection cost is linear in image size.

The render cache was shown to work well on 256x256 images using an SGI Origin 2000 with 250MHz R10k processors. At higher resolutions than 256x256, the front end has too many pixels to reproject to maintain fluidity.

2.2 Parallel render cache

Ray tracing is an irregular problem, which means that the time to compute a ray task can vary substantially depending on depth complexity. For this reason it is undesirable to run a parallel ray tracing algorithm synchronously, as this would slow down rendering of each frame to be as slow as the processor which has the most expensive set of tasks. On the other hand, synchronous operation would allow a parallel implementation of the render cache to produce exactly the same artifacts as the original render cache. We have chosen responsiveness and speed of operation over minimization of artifacts by allowing each processor to update the image asynchronously.

Our approach is to distribute the render cache functionality with the key goal of not introducing synchronization, which is analogous to frameless rendering. In our system there will be a number of renderers which will reproject point clouds and render new pixels, thereby removing the bottleneck from the original render cache implementation. Scalability is therefore assured.

We parallelize the render cache by subdividing the screen into a number of tiles. A random permutation of the list of tiles could be distributed over the processors, with each renderer managing its set of tiles independently from all other renderers. Alternatively, a global list of tiles could be maintained with each processor choosing the tile with the highest priority whenever it needs a new task to work on. While the latter option may provide better (dynamic) load balancing, we have opted for the first solution. Load balancing is achieved statically by ensuring that each processor has a sufficiently large list of tiles. The reason for choosing a static load balancing scheme has to do with memory management on the SGI Origin 3800, which is explained in more detail in Section 2.3.

Each tile has associated with it a local point cloud and an image plane data structure. The work associated with a tile depends on whether or not camera movement is detected. If the camera is moving, the point cloud is projected onto the tile's local image plane and the results are sent to the display thread for immediate display. No new rays are traced, as this would slow down the system and the perceived smoothness would be affected. This is at the cost of a degradation in image quality, which is deemed more acceptable than a loss of interaction. It is also the only modification we have applied to the render cache concept.

If there is no camera movement, a depth test is performed to select those rays that would improve image quality most. Other heuristics such as an aging scheme applied to the points in the point cloud also aid in selecting appropriate new rays. Newly traced rays are both added to the point cloud and displayed on screen. The point cloud itself does not need to be reprojected.

The renderers each loop over their allotted tiles, executing for each tile in turn the following main components:

1. **Clear tile** Before points are reprojected, the tile image is cleared.

2. **Add points** Points that previously belonged to a neighboring tile but have been projected onto the current tile are added to the point cloud.

3. **Project point cloud** The point cloud is projected onto the tile image. Points that project outside the current tile are temporarily buffered in a data structure that is periodically communicated to the relevant neighboring tiles.

4. **Depth test** A depth test is performed on the tile image to determine depth discontinuities. This is then used to select new rays to trace.

5. **Trace rays** The rays selected by the depth test function, are traced and the results added to the local point cloud.

6. **Display tile** The resulting tile is communicated to the display thread. This function also performs hole-filling to improve the image's visual appearance.

If camera movement has occurred since a tile was last visited, items 1, 2, 3 and 6 in this list are executed for that tile. If the camera was stationary, items 1, 2, 3 and 6 are executed. The algorithm is graphically depicted in Figure 2.2

While tiles can be processed largely independently, there are circumstances when interaction between tiles is necessary. This occurs for instance when a point in one tile's point cloud projects to a different tile (due to camera movement). In that case, the point is removed from the local point cloud and is inserted into the point cloud associated with the tile to which it projects. The more tiles there are, the more often this would occur. This conflicts with the goal of having many tiles for load balancing purposes. In addition, having fewer tiles that are larger causes tile boundaries to be more visible.

As each renderer produces pixels that need to be collated into an image for display on screen, there is still a display process. This display thread only displays pixels and reads the keyboard for user input. Displaying an image is achieved by reading an array of pixels that represents the entire image, and sending this array to the display hardware using OpenGL. When renderers produce pixels, they are buffered in a local data structure, until a sufficient number of pixels has been accumulated for a write into the global array of pixels. This buffering process ensures that memory contention is limited for larger image sizes.

Finally, the algorithm shows similarities with the concept of frameless rendering, in the sense that tiles are updated independently from the display process. If the size of the tiles is small with respect to the image size, the visual effect is like that of frameless rendering. The larger the tile size is chosen, the more the image updating process starts to look like a distributed version of the render cache.

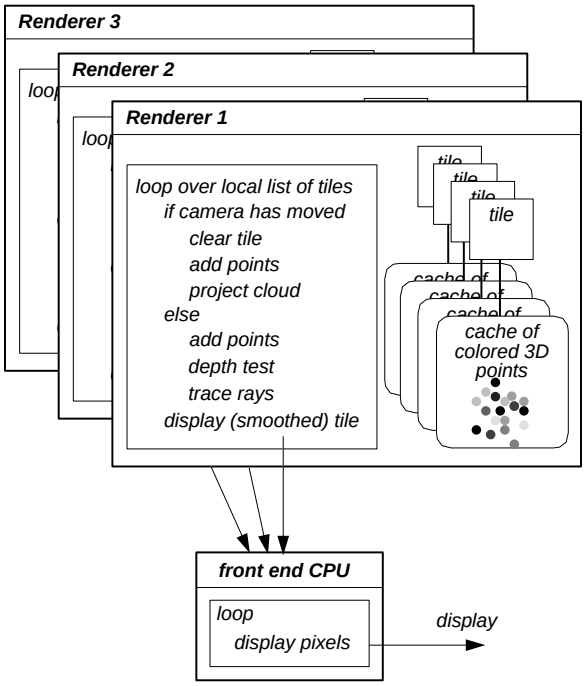


Figure 2.2: *The parallel render cache algorithm.*

2.3 Implementation details

The parallel render cache algorithm is implemented on a 32 processor SGI Origin 3800. While this machine has a 16 GB shared address space, the memory is physically distributed over a total of eight nodes. Each node features four 400 MHz R12k processors and one 2 GB block of memory. In addition each processor has an 8 MB secondary cache. Memory access times are determined by the distance between the processor and the memory that needs to be read or written. The local cache is fastest, followed by the memory associated with a processor's node. If a data item is located at a different node, fetching it may incur a substantial performance penalty.

A second issue to be addressed is that the SGI Origin 3800 may relocate a rendering process with a different processor each time a system call is performed. Whenever this happens, the data that used to be in the local cache is no longer locally available. Cache performance can thus be severely reduced by migrating processes.

These issues can be avoided on the SGI Origin 3800 by actively placing memory near the processes and disallowing process migration. This can, for example, be accomplished using the *dplace* library or the mmap system call. Associated with each tile in the parallel render cache is a local point cloud data structure and an image data structure which are mapped as close as possible to the process that uses it. Such memory mapping assures that if a cache miss occurs for any of these data structures, the performance penalty will be limited to fetching a data item that is in local memory. As argued above, this is much cheaper than fetching data from remote nodes. For this reason, using a global list of tiles as mentioned in the previous section is less efficient than distributing tiles statically over the available processors.

Carefully choreographing the mapping of processes to processors and their data structures to local memory enhances the algorithm's performance. Cache performance is improved and the number of data fetches from remote locations is minimized.

2.4 Scalability measures

The main loops of the renderers consist of a number of distinct steps. During each iteration a subset of these steps is executed dependent on whether camera movement has occurred or not (see Section 2.2). Standard speed-up measurements would under these circumstances produce unreliable results, since the measured speed-up would depend on how often the user moves the camera. The user cannot be expected to move the camera in exactly the same way for each measurement.

For this reason each of the steps making up the complete algorithm are measured separately. To assess scalability, the time to execute each step is measured, summed over all invocations and processors and subsequently divided by the number of invocations and processors. The result is expressed in events per second per processor, which for a scalable system should be independent of the number of processors employed. Hence, using more processors would then not alter the measurements. In case this measure varies with processor count, scalability is affected.

If the number of events per second per processor drops when adding processors, sublinear scalability is measured, whereas an increase indicates super-linear speed-up

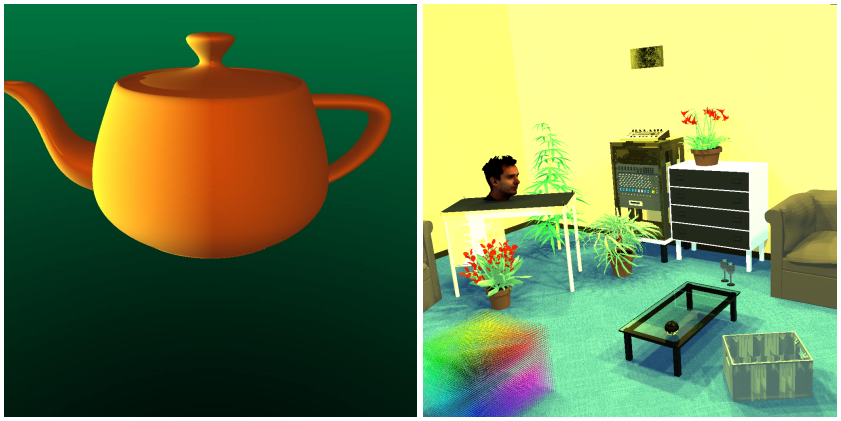


Figure 2.3: *Test scenes. The teapot (top) consists of 32 bezier patches, while the room scene consists of 846,563 primitives and 80 point light sources.*

for the measured function. Also note that the smaller the number, the more costly the operation will be. Using this measure provides better insight into the behavior of the various parts of the algorithm than a standard scalability computation would give, especially since only a subset of the components of the render cache algorithm is executed during each iteration.

2.5 Results

Our implementation uses the original render cache code of Walter et al [25][1]. Two test scenes were used: a teapot with 32 bezier patches[2] and one point light source, and a room scene with 846,563 geometric primitives and area light sources approximated by 80 point light sources (Figure 2.3). For the teapot scene, the renderer is limited by the point reprojection algorithm, while for the room scene, tracing new rays is the slowest part of the algorithm. The latter scene is of typical complexity in architectural applications and usually cannot be interactively manipulated.

In the following subsection, the different components making up the parallel render cache are evaluated (Section 2.5.1), the performance as function of task size is assessed (Section 2.5.2) and the parallel render cache is compared with other methods to speed up interactive ray tracing (Section 2.5.3).

2.5.1 Parallel render cache evaluation

The results of rendering the teapot and room models on different numbers of processors at a resolution of 512^2 and 1024^2 pixels are depicted in Figures 2.4 and 2.5.

[1]The original code has since been improved (Walter, personal communication) but we have not ported that improved code. However, we expect that any improvements to the serial code would transfer to our parallel version since the serial code runs essentially as a black box.

[2]These bezier patches are rendered directly using the intersection algorithm from Parker et. al [17].

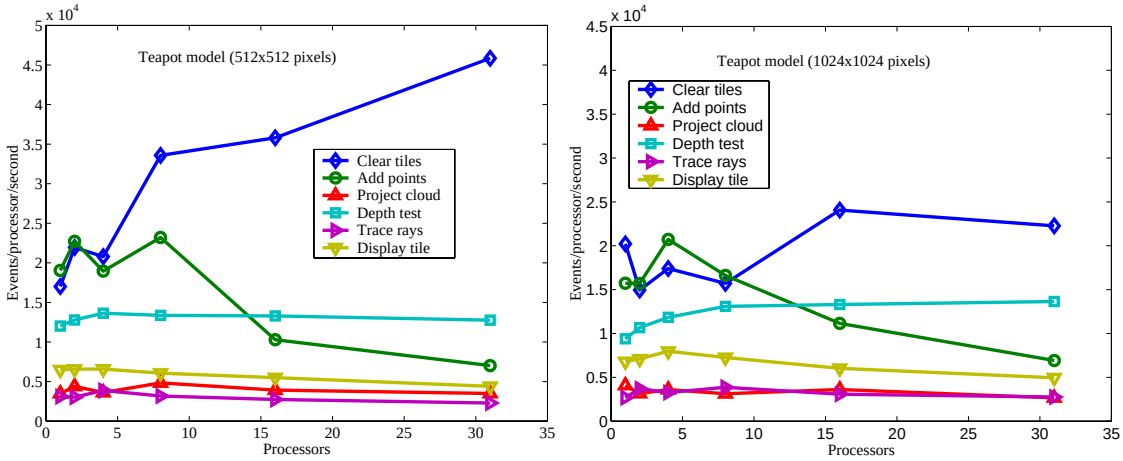


Figure 2.4: *Scalability of the render cache components for the teapot scene rendered at* 512^2 *pixels (left) and* 1024^2 *pixels (right). Negative slopes indicate sub-linear scalability, whereas horizontal lines show linear speed-ups.*

While most of the components making up the algorithm show horizontal lines in these graphs, meaning that they scale well, the "Clear tiles" and "Add point" components show non-linear behavior. Clearing tiles is a very cheap operation which appears to become cheaper if more processors are used. Because more processors result in each processor having to process fewer tiles, this super-linear behavior may be explained by better cache performance. This effect is less pronounced for the 1024^2 pixel renderings, which also points to a cache performance issue as here each processor handles more data.

The "Add point" function scales sub-linearly with the number of processors. Because the total number of tiles was kept constant between runs, this cannot be explained by assuming that different numbers of points project outside their own tile and thus have to be added to neighboring tiles. However, with more processors there is an increased probability that a neighboring tile belongs to a different processor and may therefore reside in memory which is located elsewhere in the machine. Thus projecting a point outside the tile that it used to belong to, may become more expensive for larger numbers of processors. This issue is addressed in the following section.

Note also that despite the poor scalability of "Add points", in absolute terms its cost is rather low, especially for the room model. Hence, the algorithm is bounded by components that scale well (they produce more or less horizontal lines in plots) and therefore the whole distributed render cache algorithm scales well, at least up to 31 processors (see also Section 2.5.3). In addition, the display of the results is completely decoupled from the renderers which produce new results and therefore the screen is updated at a rate that is significantly higher than rays can be traced and is also much higher than points can be reprojected. This three-tier system of producing new rays at a low frequency, projecting existing points at an intermediate frequency and display-

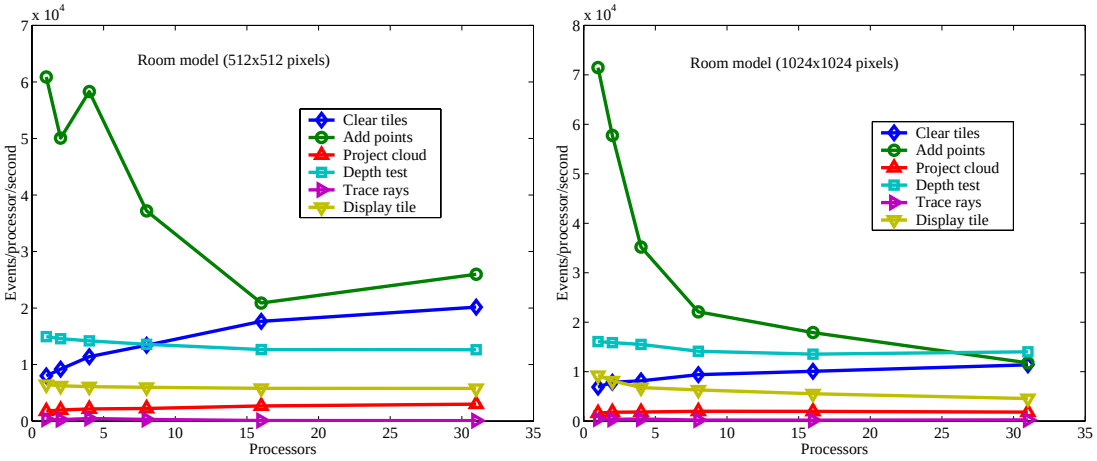


Figure 2.5: *Scalability for the room scene, rendered at* 512^2 *pixels (left) and* 1024^2 *pixels (right). Horizontal lines indicate linear scalability, whereas a fall-off means sub-linear scalability.*

ing the results at a high frequency (on the Origin 3800 at a rate of around 290 frames per second for 512^2 images and 75 frames per second for 1024^2 images, regardless of number of renderers and scene complexity) ensures a smooth display which is perceived as interactive, even if new rays are produced at a rate that would not normally allow interactivity.

By abandoning ray tracing altogether during camera movement, the system shows desirable behavior even when fewer than 31 processors are used. For both the room scene and the teapot model, the camera can move smoothly if 4 or more processors are used. During camera movement, the scene deteriorates because no new rays are produced and holes in the point cloud may become visible. During rapid camera movement, tile boundaries may become temporarily visible. After the camera has stopped moving, these artifacts disappear at a rate that is linear in the number of processors employed. We believe that maintaining fluid motion is more important than the temporary introduction of some artifacts, which is why the distributed render cache is organized as described above.

For those who would prefer a more accurate display at the cost of a slower system response, it would be possible to continue tracing rays during camera movement. Although the render cache then behaves differently, the scalability of the separate components, as given in Figures 2.4 and 2.5, would not change. However, the fluidity of camera movement is destroyed by an amount dependent on scene complexity.

2.5.2 Task size

In section 2.2 it was argued that the task size, i.e. the size of the tiles, is an important parameter which defines both speed and the occurrence of visual artifacts. The larger the task size, the better artifacts become visible. However, at the same time,

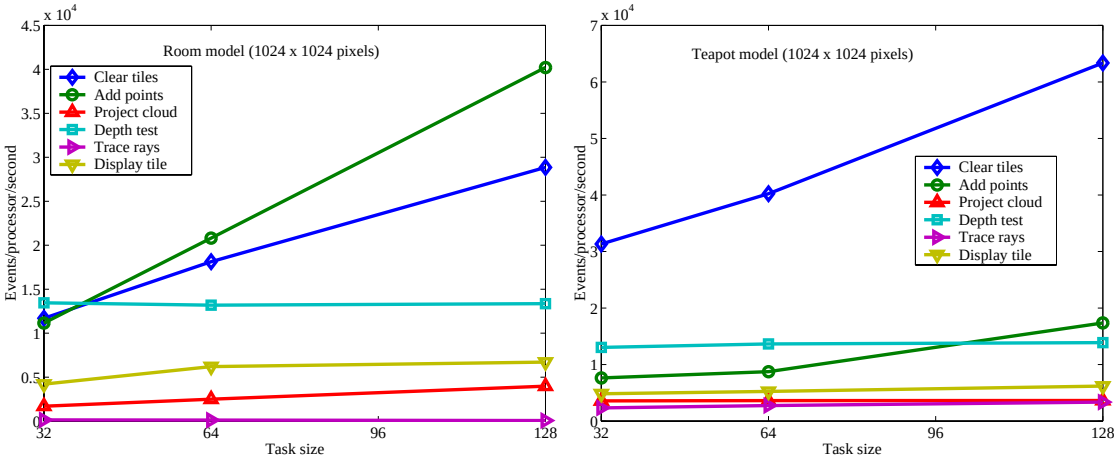


Figure 2.6: *Scalability for the room model (left) and teapot scene (right) as function of tile size (32^2, 64^2 and 128^2 pixels per tile). The image size is 1024^2 pixels and for these measurements 31 processors were used. These graphs should be interpreted the same as those in Figures 2.4 and 2.5.*

the reprojections that cross tile-boundaries are less likely to occur, resulting in higher performance. In Figure 2.6 the scalability of the parallel render cache components as function of task size is depicted. Task sizes range from 32^2 pixels to 128^2 pixels and the measurements were all obtained using 31 processors on 1024^2 images. Larger tile sizes are thus impossible, as the total number of tasks would become smaller than the number of processors. Task sizes smaller than 32^2 pixels resulted in unreasonably slow performance and were therefore left out of the assessment.

As in the previous section, the "Add points" and "Clear tile" components show interesting behavior. As expected, for larger tasks, the "Add points" function becomes cheaper. This is because the total length of the tile boundaries diminishes for larger task sizes, and so the probability of reprojections occurring across tile boundaries is smaller.

The "Clear tile" component also becomes less expensive for larger tiles. Here, we suspect that resetting one large block of memory is less expensive than resetting a number of smaller blocks of memory.

Although Figure 2.6 suggests that choosing the largest task size as possible would be appropriate, the artifacts visible for large tiles are more unsettling than for smaller task sizes. Hence, for all other experiments presented in this paper, a task size of 32^2 pixels is used, which is based on an assessment of both artifacts and performance.

2.5.3 Comparison with other speed-up mechanisms

In this section, the parallel render cache is compared with other state-of-the-art rendering techniques. All make use of the interactive ray tracer of Parker et. al. [17], either as a back-end or as the main algorithm. The comparison includes the original render

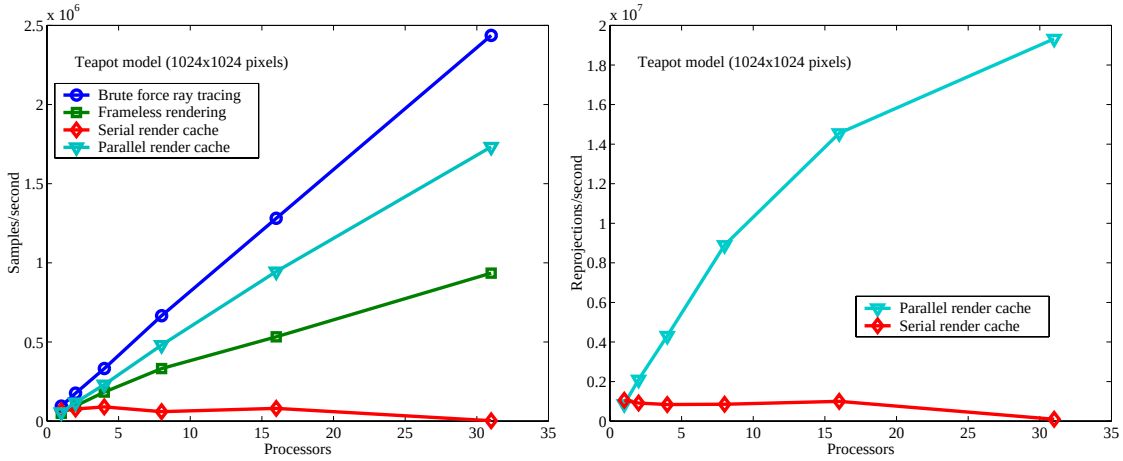


Figure 2.7: *Samples per second (left) and point reprojections per second (right) for the teapot model.*

cache algorithm [25], the parallel render cache algorithm as described in this paper, the interactive ray tracer (rtrt) without reprojection techniques and the interactive ray tracer using the frameless rendering concept [17]. In the following we will refer to the original render cache as "serial render cache" to distinguish it from our parallel render cache implementation. All renderings were made using the teapot and room models (Figure 2.3) at a resolution of 1024^2 pixels.

The measurements presented in this section consist of the number of new samples produced per second by each of the systems and the number of points reprojected per second (for the two render cache algorithms). These numbers are summed over all processors and should therefore scale with the number of processors employed. The results for the teapot model are given in Figure 2.7 and the results for the room model are presented in Figure 2.8.

The graphs on the left of these figures show the number of samples generated per second. All the lines are straight, indicating scalable behavior. In these plots, steeper lines are the result of higher efficiency and therefore, the real-time ray tracer would be most efficient, followed by the parallel render cache. The frameless rendering concept looses efficiency because randomizing the order in which pixels are generated destroys cache coherence. The parallel render cache does not suffer from this, since the screen is tiled and tasks are based on tiles. The serial render cache appears to perform well for complex scenes and poorly for simple scenes. For scenes that lack complexity, the point reprojection front-end becomes the bottleneck, especially since the image size chosen causes the point cloud to be quite large. Thus, the render cache front-end needs to reproject a large number of points for each frame and so constitutes a bottleneck.

Although the parallel render cache does not produce as many new pixels as the real-time ray tracer by itself does, this loss of efficiency is compensated by its ability to reproject large numbers of points, as is shown in the plots on the right of Figures 2.7

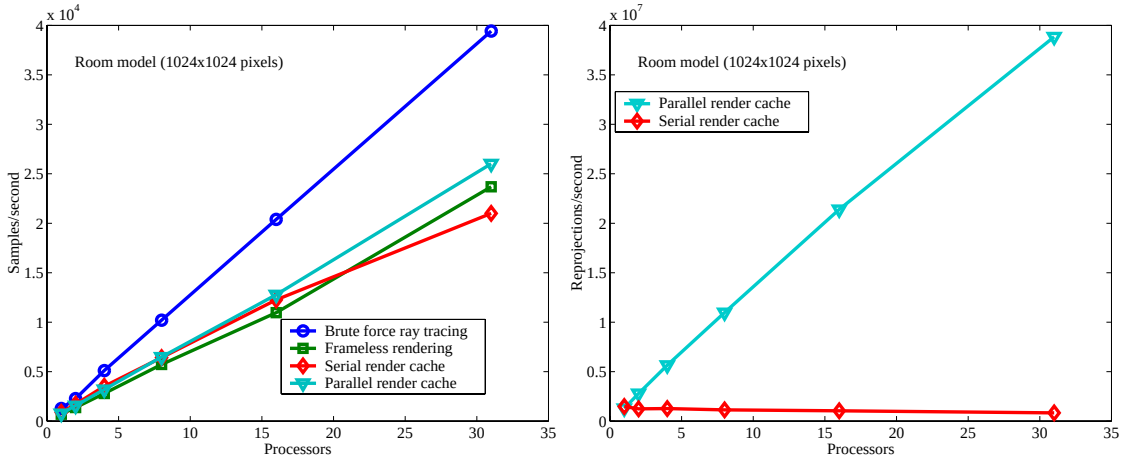


Figure 2.8: *Samples per second (left) and point reprojections per second (right) for the room scene.*

and 2.8. The point reprojection component of the parallel render cache shows good scalability, and therefore the goal of parallelizing the render cache algorithm is reached. The point reprojection part of the serial render cache does not scale because it is serial in nature.

2.6 Discussion

While it is true that processors get ever faster and multi-processor machines are now capable of real-time ray tracing, scenes are getting more and more complex while at the same time frame sizes still need to increase. Hence, Moore's law is not likely to allow interactive full-screen brute-force ray tracing of highly complex scenes anytime soon.

Interactive manipulation of complex models is still not possible without the use of sophisticated algorithms that can efficiently exploit temporal coherence. The render cache is one such algorithm that can achieve this. However, for it not to become a bottleneck itself, the render cache functionality needs to be distributed over the processors that produce new samples. The resulting algorithm, presented in this paper, shows superior reprojection capabilities that enables smooth camera movement, even in the case where the available processing power is much lower than would be required in a brute force approach. It achieves this for scene complexities and image resolutions that are not feasible using any of the other algorithms mentioned in the previous section.

While smoothness of movement is an important visual cue, our algorithm necessarily produces other artifacts during camera motion. These artifacts are deemed less disturbing than jerky motion and slow response times. The render cache attempts to fill small holes after point reprojection. For larger holes, this may fail and unfilled pixels may either be painted in a fixed color, or can be left unchanged from previous repro-

jections. Either approach causes artifacts inherent to the algorithm and is present both in the original render cache and in our parallel implementation of it.

The parallel render cache produces additional artifacts due to the tiling scheme employed. During camera movement, tile boundaries may temporarily become visible, because there is some latency between points being reprojected from neighboring tiles and this reprojection becoming visible in the current tile. A further investigation to minimize these artifacts is in order, which we reserve for future work. Currently, the parallel render cache algorithm is well suited for navigation through highly complex scenes to find appropriate camera positions.

It has been shown that even with a relatively modest number of processors, the distributed render cache can produce smooth camera movement at resolutions typically sixteen times higher than the original render cache. The system as presented here scales well up to 31 processors. Its linear behavior suggests that improved performance is likely beyond 31 processors, although if this many processors are available, it would probably become sensible to devote the extra processing power to produce more samples, rather than increase the speed of reprojection.

3 Animation

A fully optimized ray tracer which allows interactive walk-throughs is attractive over other interactive rendering algorithms because it allows a large set of effects to be rendered which are more difficult or even impossible to obtain using graphics hardware. In addition, ray tracing scales sub-linearly in the number of objects due to the use of spatial subdivisions. It also scales sub-linearly in the number of pixels rendered, provided cache coherency can be exploited fully.

To make interactive ray tracing more attractive, we have looked into ways to enable objects to be manipulated in real-time ([20], reproduced with permission in Appendix A). In the following we assume that animation paths are not known prior to the rendering, and so updates to the scene need to be achieved in real-time with as little overhead as possible. In addition it is important that the effect of time-varying scenes on the performance of the renderer is as small as possible.

Changing the coordinates of an object in real-time is not particularly difficult to achieve, so we will not address this issue in any detail. However, an object's change in location, size or rotation does imply that after the transformation, the object may occupy a different portion of space. In the absence of a spatial subdivision to speed up the intersection tests, this would not constitute a problem.

However, the current speed of the hardware, combined with the number of computations required to ray trace an image, does not allow us to do away with spatial subdivisions altogether. Additionally, spatial subdivisions are usually built as a pre-process to rendering. The cost of building a spatial subdivision is not negligible. Hence, spatial subdivisions are required to obtain interactive frame-rates, but at the same time they are not flexible enough to accommodate time-varying data.

In this section we describe a simple adaptation to both grid and octree spatial subdivisions which caters for a small number of animated objects. These objects can either be animated according to pre-defined motion splines or they can be picked up by the user and placed elsewhere in the scene. Animating all objects at the same time in a complex scene is not yet possible. It would require rebuilding the entire spatial subdivision for each frame and this is too costly to achieve using current technology. Focusing on just a small number of objects to be animated/manipulated allows the design of spatial subdivisions which can be incrementally updated after each frame.

In the following sub-sections the basic idea is explained (Section 3.1) and results are shown (Section 3.2). We would also like to refer to Appendix A which includes a full publication regarding this subject. The results presented in this chapter are obtained using an SGI Origin 3800, while appendix A contains older results using an SGI Origin

2000.

3.1 Algorithm

In this section modifications to grid and octree spatial subdivisions are discussed. The octree is a hierarchical extension to the grid. We assume the reader is familiar with these spatial subdivisions [1, 5, 7, 9, 11, 13, 15, 16, 24, 26].

Grid spatial subdivisions for static scenes, without any modifications, are already useful for animated scenes, as traversal costs are low and insertion and deletion of objects is reasonably straightforward. Insertion and deletion are considered basic operations necessary for the animation of objects. The general approach is to remove an object from the spatial subdivision, modify its coordinates and the re-insert the object into the acceleration structure. Insertion is usually accomplished by mapping the axis-aligned bounding box of an object to the voxels of the grid. The object is inserted into all voxels that overlap with this bounding box. Deletion can be achieved in a similar way.

However, when an object moves outside the extent of the spatial subdivision, the acceleration structure would normally have to be rebuilt. As this is too expensive to perform repeatedly, we propose to logically replicate the grid over space. If an object exceeds the bounds of the grid, the object wraps around before re-insertion. Ray traversal then also wraps around the grid when a boundary is reached. In order to provide a stopping criterion for ray traversal, a logical bounding box is maintained which contains all objects, including the ones that have crossed the original perimeter. As this scheme does not require grid re-computation whenever an object moves far away, the cost of maintaining the spatial subdivision will be substantially lower. On the other hand, because rays now may have to wrap around, more voxels may have to be traversed per ray, which will slightly increase ray traversal time.

During a pre-processing step, the grid is built as usual. We will call the bounding box of the entire scene at start-up the 'physical bounding box'. If during the animation an object moves outside the physical bounding box, either because it is placed by the user in a new location, or its programmed path takes it outside, the logical bounding box is extended to enclose all objects. Initially, the logical bounding box is equal to the physical bounding box. Insertion of an object which lies outside the physical bounding box is accomplished by wrapping the object around within the physical grid, as depicted in Figure 3.1 (left).

As the logical bounding box may be larger than the physical bounding box, ray traversal now starts at the extended bounding box and ends if an intersection is found or if the ray leaves the logical bounding box. In the example in Figure 3.1 (right), the ray pointing to the sphere starts within a logical voxel, voxel (0, -2), which is mapped to physical voxel (0, 2). The logical coordinates of the sphere are checked and found to be outside of the currently traversed voxel and thus no intersection test is necessary. The ray then progresses to physical voxel (1, 2). For the same reason, no intersection with the sphere is computed again. Traversal then continues until the sphere is intersected in logical voxel (4, 2), which maps to physical voxel (0, 2).

Objects that are outside the physical grid are tagged, so that in the above example,

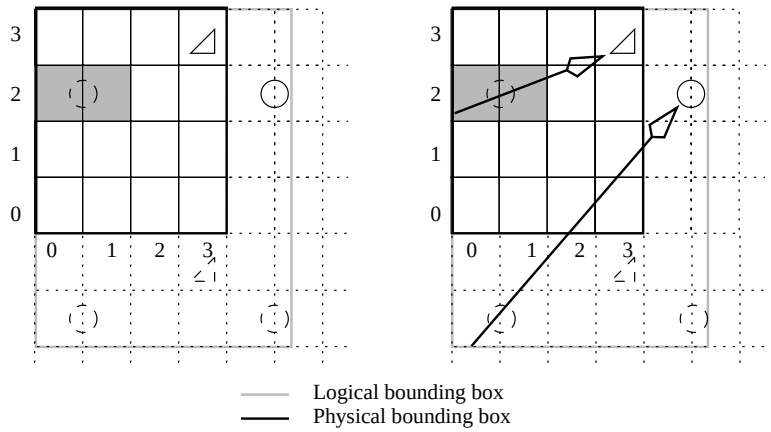


Figure 3.1: Grid insertion (left). The sphere has moved outside the physical grid, now overlapping with voxels (4, 2) and (5, 2). Therefore, the object is inserted at the location of the shaded voxels. The logical bounding box is extended to include the newly moved object. Right: ray traversal through extended grid. The solid lines are the actual objects whereas the dashed lines indicate voxels which contain objects whose actual extents are not contained in that voxel.

when the ray aimed at the triangle enters voxels (0, 2) and (1, 2), the sphere does not have to be intersected. Similarly, when the ray is outside the physical grid, objects that are within the physical grid need not be intersected. As most objects will initially lie within the physical bounds, and only a few objects typically move away from their original positions, this scheme speeds up traversal considerably for parts of the ray that are outside the physical bounding box.

When the logical bounding box becomes much larger than the physical bounding box, there is a tradeoff between traversal speed (which deteriorates for large logical bounding boxes) and the cost of rebuilding the grid. In our implementation, the grid is rebuilt when the length of the diagonals of the physical and logical bounding boxes differ by a factor of two. This heuristic aims to provide a trade-off between traversal speed and the frequency with which the spatial subdivision needs to be re-generated.

Hence, there is a hierarchy of operations that can be performed on grids. For small to moderate expansions of the scene, wrapping both rays and objects is relatively quick without incurring too high a traversal cost. For larger expansions, rebuilding the grid will become a more viable option.

This grid implementation shares the advantages of simplicity and cheap traversal with commonly used grid implementations. However, it adds the possibility of increasing the size of the scene without having to completely rebuild the grid every time there is a small change in scene extent.

The cost of deleting and inserting a single object is not constant and depends largely on the size of the object relative to the size of the scene. The size of an object relative to each voxel in a grid influences how many voxels will contain that object. This in turn negatively affects insertion and deletion times. Hence, it would make sense to find a

spatial subdivision whereby the voxels can have different sizes. If this is accomplished, then insertion and deletion of objects can be made independent of their sizes and can therefore be executed in constant time. Such spatial subdivisions are not new and are known as hierarchical spatial subdivisions. Octrees, bintrees and hierarchical grids are all examples of hierarchical spatial subdivisions. However, normally such spatial subdivisions store all their objects in leaf nodes and would therefore still incur non-constant insertion and deletion costs. We extend the use of hierarchical grids in such a way that objects can also reside in intermediary nodes or even in the root node for objects that are nearly as big as the entire scene.

Because such a structure should also be able to deal with expanding scenes, our efforts were directed towards constructing a hierarchy of grids (similar to Sung [24]), thereby extending the functionality of the grid structure presented in the previous section. Effectively, the proposed method constitutes a balanced octree.

Object insertion now proceeds similarly to grid insertion, except that the grid level needs to be determined before insertion. This is accomplished by comparing the size of the object in relation to the size of the scene. A simple heuristic is to determine the grid level from the diagonals of the two bounding boxes. Specifically, the length of the grid's diagonal is divided by the length of the object's diagonal, the result determining the grid level. Insertion and deletion progresses as explained above.

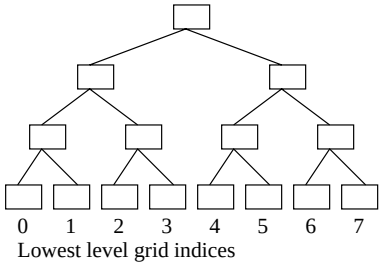
The gain of better control over insertion time is offset by a slightly more complicated traversal algorithm. Hierarchical grid traversal is effectively the same as grid traversal with the following modifications. Traversal always starts at a leaf node which may first be mapped to a physical leaf node as described earlier in this section. The ray is intersected with this voxel and all its parents until the root node is reached. This is necessary because objects at all levels in the hierarchy may occupy the same space as the currently traversed leaf node. If an intersection is found within the space of the leaf node, then traversal is finished. If not, the next leaf node is selected and the process is repeated.

This traversal scheme is wasteful because the same parent nodes may be repeatedly traversed for the same ray. To combat this problem, note that common ancestors of the current leaf node and the previously intersected leaf node, need not be traversed again (Figure 3.2). If the ray direction is positive, the current voxel's number can be used to derive the number of levels to go up in the tree to find the common ancestor between the current and the previously visited voxel. For negative ray directions, the previously visited voxel's number is used instead. Finding the common ancestor is achieved using simple bit manipulation, as detailed in Figure 3.3.

As the highest levels of the grid may not contain any objects, ascending all the way to the highest level in the grid is not always necessary. Ascending the tree for a particular leaf node can stop when the largest voxel containing objects is visited.

This hierarchical grid structure has the following features. The traversal is only marginally more complex than standard grid traversal. In addition, wrapping of objects in the face of expanding scenes is still possible. If all objects are the same size, this algorithm effectively defaults to grid traversal. Insertion and deletion times are much better controlled than for the interactive grid[1].

[1]Note that this also obviates the need for mailbox systems to avoid redundant intersection tests.



Grid Index	Levels between leaf and previously checked common ancestor
0	4 (root)
1	1
2	2
3	1
4	3
5	1
6	2
7	1

Figure 3.2: Hierarchical grid traversal. Assuming that ray traversal starts at node 0 and goes in positive direction, then after each step, the common ancestor is found *n* levels above the leaf node as indicated in the table.

```
bitmask = (raydir_x > 0) ? x : x + 1
forall levels in hierarchical grid
{
  cell = hgrid[level][x>>level][y>>level][z>>level]
  forall objects in cell
    intersect(ray, object)
  if (bitmask & 1)
    return
  bitmask >>= 1
}
```

Figure 3.3: Hierarchical grid traversal algorithm in C-like pseudo-code. The bitmask is set assuming that the last step was along the x-axis.

3.2 Results

The grid and hierarchical grid spatial subdivisions were implemented using an interactive ray tracer [17], which runs on an SGI Origin 3800 with 32 processors and 16GB of main memory[2]. Each processor is an R12k running at 400Mhz and manages an 8MB secondary cache. We have chosen to use 30 processors for rendering and one extra thread to take care of user input, displaying the frames, and also for updating and rebuilding the spatial subdivision when necessary (one processor remained unused by our application to allow for system processes to run smoothly). The reason to include the scene update routines with the display thread is that querying the keyboard and displaying the images takes very little time. The remainder of the time to calculate a frame could therefore easily be spent animating objects. In addition, it is important that the scene updates are completed within the time to compute a new frame, as longer update times would either cause delays or result in jerky movement of objects. As the frame rate depends on both the scene complexity and the number of processors that participate in the calculation, the time to update the scene is dependent on both of these parameters.

For evaluation purposes, two test scenes were used. In each scene, a number of

[2]Note that the original work, presented in Appendix A, reported results obtained on a slower Origin 2000.

objects were animated using pre-programmed motion paths. The scenes as they are at start-up are depicted in Figure A.5 (top, Appendix A). An example frame taken during the animation is given for each scene in Figure A.5 (bottom, Appendix A). All images were rendered at a resolution of 512^2 pixels.

3.2.1 Traversal performance - static scenes

The performance penalty incurred by the new grid and hierarchical grid implementations are assessed by comparing these with a standard grid implementation. The standard grid data structure consists of a single array of object pointers. This design allows better cache efficiency on the SGI Origin series. Finally, we have also implemented a hierarchical grid with a higher branching factor. Instead of subdividing a voxel into eight children, here nodes are split into 64 children (4 along each axis).

From here on we will refer to the new grid implementation as 'interactive grid' to distinguish between the two grid traversal algorithms. As all these spatial subdivision methods have a user defined parameter to set the resolution (voxels along one axis and maximum number of grid levels, respectively), various settings are evaluated. The overall performance is given in Figure 3.4 and is measured in frames per second.

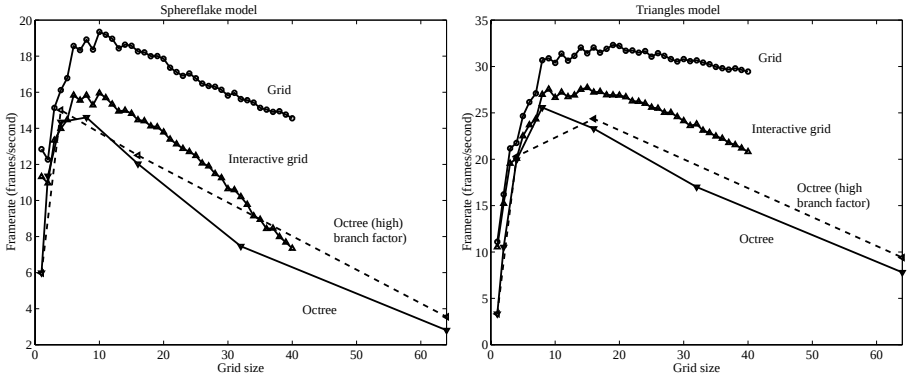


Figure 3.4: Performance (in frames per second) for the grid, the interactive grid and the hierarchical grids for two static scenes.

The extra flexibility gained by both the interactive grid and hierarchical grid implementations results in a somewhat slower frame rate. This is according to expectation, as the traversal algorithm is a little more complex and the Origin's cache structure cannot be exploited as well with either of the new grid structures. The graphs in Figure 3.4 should be compared to our previous results given in Figure A.3 in Appendix A. For the hierarchical grid with the higher branching factor, the observed frame rates are very similar to the hierarchical grid.

3.2.2 Object update rate - dynamic scenes

The object update rates were slightly better for the sphereflake and triangle scenes, because the size differences between the objects matches this acceleration structure better than both the interactive grid and the hierarchical grid.

The non-zero cost of updating the scene effectively limits the number of objects that can be animated within the time-span of a single frame. However, for both scenes, this limit was not reached. For each of these tests, the hierarchical grid is more efficiently updated than the interactive grid, which confirms its usefulness.

The size difference between different objects should cause the update efficiency to be variable for the interactive grid, while remaining relatively constant for the hierarchical grid. In order to demonstrate this effect, both the ground plane and one of the triangles in the triangle scene was interactively repositioned during rendering. Similarly, in the sphereflake scene one of the large spheres and one of the small spheres were interactively manipulated. The update rates for different size parameters for both the interactive grid and the hierarchical grid, are presented in Figure 3.5. Comparing the grid size of 16 for the interactive grid with the size parameter of 4 for the interactive grid in this figure, shows that for similar numbers of voxels (at the deepest level of the hierarchical grid) along each axis, the update rate varies much more dependent on object size for the interactive grid than for the hierarchical grid. Hence, the hierarchical grid copes much better with objects of different sizes than the interactive grid. Dependent on the number of voxels in the grid, there is one to two orders of magnitude difference between inserting a large and a small object. For larger grid sizes, the update time for the ground plane of the triangles scene is roughly half a frame. This leads to visible artifacts when using the interactive grid, as during the update the processors that are rendering the next frame temporarily cannot intersect this object (it is simply taken out of the spatial subdivision). In practice, the hierarchical grid implementation does not show this disadvantage.

The time to rebuild a spatial subdivision from scratch is expected to be considerably higher than the cost of re-inserting a small number of objects. For the triangles scene, where 200 out of 201 objects were animated, the update rate was still a factor of two faster than the cost of completely rebuilding the spatial subdivision. This was true for both the interactive grid and the hierarchical grid. A factor of two was also found for the animation of 81 spheres in the sphereflake scene. When animating only 9 objects in this scene, the difference was a factor of 10 in favor of updating. We believe that the performance difference between rebuilding the acceleration structure and updating all objects is largely due to the cost of memory allocation, which occurs when rebuilding. The cost of rebuilding the spatial subdivision will become prohibitive when much larger scenes are rendered.

3.2.3 Traversal cost - dynamic scenes

In the case of expanding scenes, the logical bounding box will become larger than the physical bounding box. The number of voxels that are traversed per ray will therefore on average increase. This is the case in the triangles scene. The variation over time of

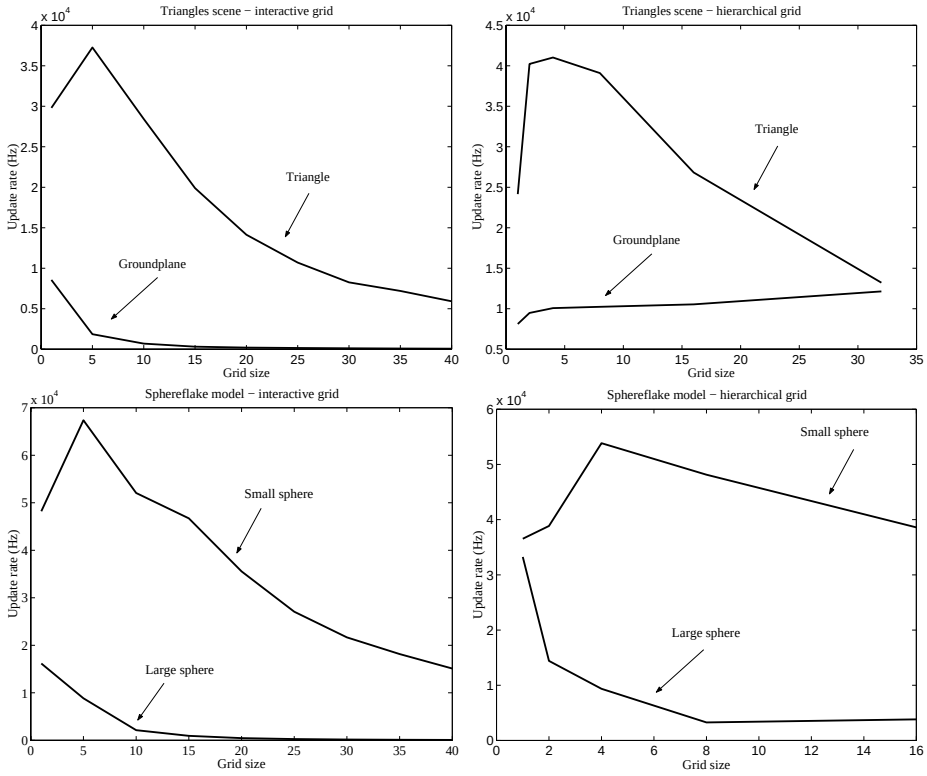


Figure 3.5: Update rate as function of grid size for the interactive and hierarchical grids.. We compare the update rates for a small as well as a large object in both the triangles model (top) and the sphereflake model (bottom).

the frame rate is given in Figure 3.6. In this example, the objects are first stationary. At some point the animation starts and the frame rate drops because the scene immediately starts expanding. For the sphereflake scene, the animated objects do not cause the scene to expand, and therefore no drop in framerate is observed.

3.2.4 Animating clusters of objects

For many applications it will be necessary to animate clusters of objects in a coherent manner. For example, if a teapot such as depicted in Figure 3.7, needs to be repositioned, it would not make sense to individually move each of its 25,000 individual polygons. Encapsulating the teapot within its own spatial subdivision will improve rendering time but will not improve insertion and deletion time, as after moving the teapot, all its polygons would still require updating. Here, the use of instancing provides a good solution as only the transformation matrix specifying where the teapot is positioned in space will need to be updated. For this example, updating the spatial subdivision as well as the transformation matrix can be performed around 2400 times

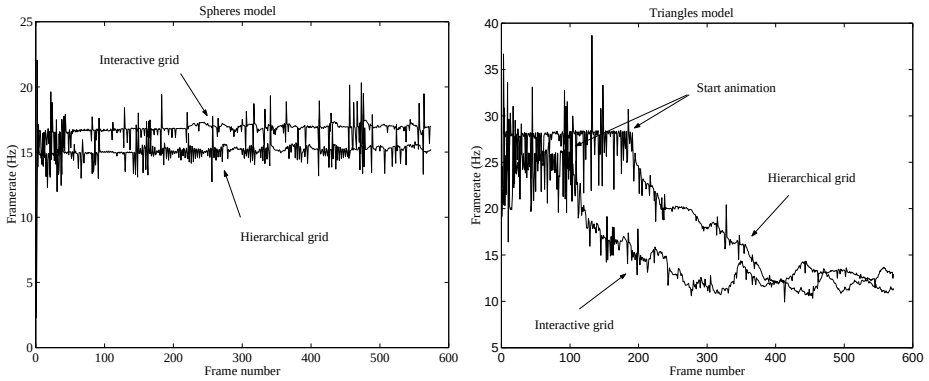


Figure 3.6: Framerate as function of time for the triangles scene and the sphereflake scene. Note that the sphereflake scene does not expand over time and therefore starting the animation does not appreciably affect the framerate.

per second. The benchmark for this scene resulted in a frame rate of 12.1 fps[3].

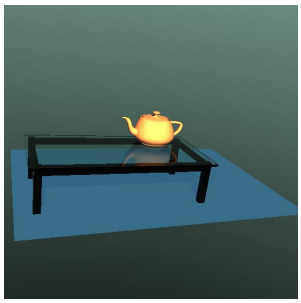


Figure 3.7: Example of instancing. Moving the teapot requires a cheap update of a transformation matrix.

Finally, Figure A.6 shows that interactively updating scenes using drag and drop interaction is feasible.

3.3 Discussion

When objects are interactively manipulated and animated within a ray tracing application, much of the work that is traditionally performed during a pre-processing step becomes a limiting factor. Especially spatial subdivisions which are normally built once before the computation starts, do not exhibit the flexibility that is required for animation. The insertion and deletion costs can be both unpredictable and variable. We have argued that for a small cost in traversal performance flexibility can be obtained

[3]Result obtained on a 32-node SGI Origin 2000.

and insertion and deletion of objects can be performed in a well controlled amount of time.

By logically extending the (hierarchical) grids into space, these spatial subdivisions deal with expanding scenes rather naturally. For modest expansions, this does not significantly alter the frame rate. When the scenes expand a great deal, rebuilding the entire spatial subdivision may become necessary. For large scenes this may involve a temporary drop in frame rate. For applications where this is unacceptable, it would be advisable to perform the rebuilding within a separate thread (rather than the display thread) and use double buffering of the scene to minimize the impact on the rendering threads.

4 Summary and discussion

In chapter 2 we have described algorithmic extensions to an interactive ray tracer that can be employed to allow much larger images to be computed or have scenes of much higher complexity rendered interactively. Point reprojection techniques do not allow rays to be produced quicker, but rely on temporal coherence by reusing samples computed for previous frames. For complex scenes, reprojection of existing results is much faster than tracing new rays and so point reprojection allows for smooth movement between camera points for scenes that are too complex for other algorithms to smoothly advance from one camera position to the next.

In chapter 3 extensions to spatial subdivisions were discussed. Normally, spatial acceleration structures are built as a preprocess and are therefore not flexible enough to accomodate interactively placed or moved objects. Extending grid and octree data structures to enable user interaction with the scene interactively have a modest impact on speed of rendering which is acceptable given their ability to allow objects to be moved within and even outside the extent of the scene.

Interactive ray tracing is now feasible and for certain types of application, such as interactive rendering of the visible female data set [18], it is a better choice than other forms of rendering, including z-buffer techniques. With increasing available computational power, the range of applications for interactive ray tracing is likely to grow and become possible on cheaper hardware.

Acknowledgements

We would like to thank Brian Smits, Chuck Hansen and Pete Shirley for their help, support and involvement in the projects that are described in these course notes. In addition, we thank Bruce Walter, Steven Parker and George Drettakis for their render cache source code and John McCorquodale for valuable discussions regarding processor and memory placement issues. Thanks also to Springer Verlag for allowing us to republish appendix A. This work was supported by NSF grants NSF/ACR, NSF/MRI and by the DOE AVTC/VIEWS.

Appendix

Appendix A contains the following paper, which is reproduced with permission from Springer Wien - New York:

> Reinhard, E., Smits, B., Hansen, C., *Dynamic Acceleration Structures for Interactive Ray Tracing*. in: Rendering Techniques 2000 (Proceedings of the Eurographics Workshop in Brno, Czech Republic, 2000). pp299-306. Wien - New York: Springer. 2000

A Dynamic Acceleration Structures for Interactive Ray Tracing

Acceleration structures used for ray tracing have been designed and optimized for efficient traversal of static scenes. As it becomes feasible to do interactive ray tracing of moving objects, new requirements are posed upon the acceleration structures. Dynamic environments require rapid updates to the acceleration structures. In this paper we propose spatial subdivisions which allow insertion and deletion of objects in constant time at an arbitrary position, allowing scenes to be interactively animated and modified.

A.1 Introduction

Recently, interactive ray tracing has become a reality [14, 17], allowing exploration of scenes rendered with higher quality shading than with traditional interactive rendering algorithms. A high frame-rate is obtained through parallelism, using a multiprocessor shared memory machine. This approach has advantages over hardware accelerated interactive systems in that a software-based ray tracer is more easily modified. One of the problems with interactive ray tracing is that previous implementations only dealt with static scenes or scenes with a small number of specially handled moving objects. The reason for this limitation is that the acceleration structures used to make ray tracing efficient rely on a significant amount of preprocessing to build. This effectively limits the usefulness of interactive ray tracing to applications which allow changes in camera position. The work presented in this paper is aimed at extending the functionality of interactive ray tracing to include applications where objects need to be animated or interactively manipulated.

When objects can freely move through the scene, either through user interaction, or due to system-determined motion, it becomes necessary to adapt the acceleration methods to cope with changing geometry. Current spatial subdivisions tend to be highly optimized for efficient traversal, but are difficult to update quickly for changing geometry. For static scenes this suffices, as the spatial subdivision is generally constructed during a pre-processing step. However, in animated scenes pre-processed spatial subdivisions may have to be recalculated for each change of the moving objects. One approach to

circumvent this issue is to use 4D radiance interpolants to speed-up ray traversal [2]. However, within this method the frame update rates depend on the type of scene edits performed as well as the extent of camera movement. We will therefore focus on adapting current spatial subdivision techniques to avoid these problems.

To animate objects while using a spatial subdivision, insertion and deletion costs are not negligible, as these operations may have to be performed many times during rendering. In this paper, spatial subdivisions are proposed which allow efficient ray traversal as well as rapid insertion and deletion for scenes where the extent of the scene grows over time.

The following section presents a brief overview of current spatial subdivision techniques (Section A.2), followed by an explanation of our (hierarchical) grid modifications (Sections A.3 and A.4). A performance evaluation is given in Section A.5, while conclusions are drawn in the final section.

A.2 Acceleration Structures for Ray Tracing

There has been a great deal of work done on acceleration structures for ray tracing [11]. However, little work has focused on ray tracing moving objects. Glassner presented an approach for building acceleration structures for animation [10]. However, this approach does not work for environments without *a priori* knowledge of the animation path for each object. In a survey of acceleration techniques, Gaede and Günther provide an overview of many spatial subdivisions, along with the requirements for various applications [8]. The most important requirements for ray tracing are fast ray traversal and adaptation to unevenly distributed data. Currently popular spatial subdivisions can be broadly categorized into bounding volume hierarchies and voxel based structures.

Bounding volume hierarchies create a tree, with each object stored in a single node. In theory, the tree structure allows $O(\log n)$ insertion and deletion, which may be fast enough. However, to make the traversal efficient, the tree is augmented with extra data, and occasionally flattened into an array representation [22], which enables fast traversal but insertion or deletion incur a non-trivial cost. Another problem is that as objects are inserted and deleted, the tree structure could become arbitrarily inefficient unless some sort of rebalancing step is performed as well.

Voxel based structures are either grids [1, 7] or can be hierarchical in nature, such as bintrees and octrees [9, 23]. The cost of building a spatial subdivision tends to be $O(n)$ in the number of objects. This is true for both grids and octrees. In addition, the cost of inserting a single object may depend on its relative size. A large object generally intersects many voxels, and therefore incurs a higher insertion cost than smaller objects. This can be alleviated through the use of modified hierarchical grids, as explained in Section A.4. The larger problem with spatial subdivision approaches is that the grid structure is built within volume bounds that are fixed before construction. Although insertion and deletion may be relatively fast for most objects, if an object is moved outside the extent of the spatial subdivision, current structures would require a complete rebuild. This problem is addressed in the next section.

A.3 Grids

Grid spatial subdivisions for static scenes, without any modifications, are already useful for animated scenes, as traversal costs are low and insertion and deletion of objects is reasonably straightforward. Insertion is usually accomplished by mapping the axis-aligned bounding box of an object to the voxels of the grid. The object is inserted into all voxels that overlap with this bounding box. Deletion can be achieved in a similar way.

However, when an object moves outside the extent of the spatial subdivision, the acceleration structure would normally have to be rebuilt. As this is too expensive to perform repeatedly, we propose to logically replicate the grid over space. If an object exceeds the bounds of the grid, the object wraps around before re-insertion. Ray traversal then also wraps around the grid when a boundary is reached. In order to provide a stopping criterion for ray traversal, a logical bounding box is maintained which contains all objects, including the ones that have crossed the original perimeter. As this scheme does not require grid re-computation whenever an object moves far away, the cost of maintaining the spatial subdivision will be substantially lower. On the other hand, because rays now may have to wrap around, more voxels may have to be traversed per ray, which will slightly increase ray traversal time.

During a pre-processing step, the grid is built as usual. We will call the bounding box of the entire scene at start-up the 'physical bounding box'. If during the animation an object moves outside the physical bounding box, either because it is placed by the user in a new location, or its programmed path takes it outside, the logical bounding box is extended to enclose all objects. Initially, the logical bounding box is equal to the physical bounding box. Insertion of an object which lies outside the physical bounding box is accomplished by wrapping the object around within the physical grid, as depicted in Figure A.1 (left).

As the logical bounding box may be larger than the physical bounding box, ray traversal now starts at the extended bounding box and ends if an intersection is found or if the ray leaves the logical bounding box. In the example in Figure A.1 (right), the ray pointing to the sphere starts within a logical voxel, voxel (-2, 0), which is mapped to physical voxel (0, 2). The logical coordinates of the sphere are checked and found to be outside of the currently traversed voxel and thus no intersection test is necessary. The ray then progresses to physical voxel (1, 2). For the same reason, no intersection with the sphere is computed again. Traversal then continues until the sphere is intersected in logical voxel (4, 2), which maps to physical voxel (0, 2).

Objects that are outside the physical grid are tagged, so that in the above example, when the ray aimed at the triangle enters voxels (0, 2) and (1, 2), the sphere does not have to be intersected. Similarly, when the ray is outside the physical grid, objects that are within the physical grid need not be intersected. As most objects will initially lie within the physical bounds, and only a few objects typically move away from their original positions, this scheme speeds up traversal considerably for parts of the ray that are outside the physical bounding box.

When the logical bounding box becomes much larger than the physical bounding box, there is a tradeoff between traversal speed (which deteriorates for large logical bounding boxes) and the cost of rebuilding the grid. In our implementation, the grid

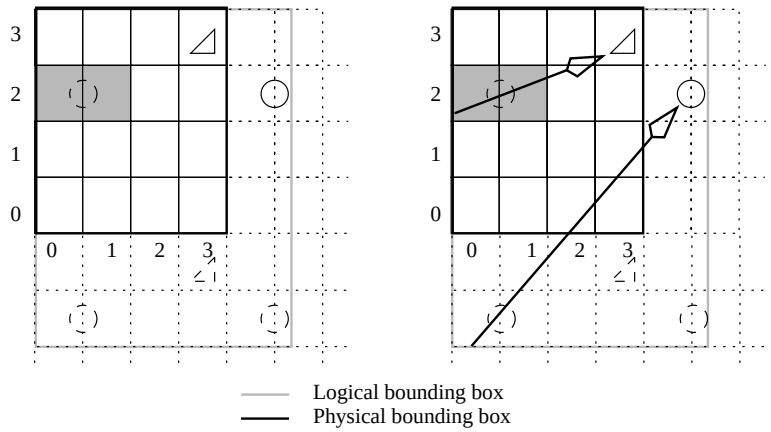


Figure A.1: Grid insertion (left). The sphere has moved outside the physical grid, now overlapping with voxels (4, 2) and (5, 2). Therefore, the object is inserted at the location of the shaded voxels. The logical bounding box is extended to include the newly moved object. Right: ray traversal through extended grid. The solid lines are the actual objects whereas the dashed lines indicate voxels which contain objects whose actual extents are not contained in that voxel.

is rebuilt when the length of the diagonals of the physical and logical bounding boxes differ by a factor of two.

Hence, there is a hierarchy of operations that can be performed on grids. For small to moderate expansions of the scene, wrapping both rays and objects is relatively quick without incurring too high a traversal cost. For larger expansions, rebuilding the grid will become a more viable option.

This grid implementation shares the advantages of simplicity and cheap traversal with commonly used grid implementations. However, it adds the possibility of increasing the size of the scene without having to completely rebuild the grid every time there is a small change in scene extent. The cost of deleting and inserting a single object is not constant and depends largely on the size of the object relative to the size of the scene. This issue is addressed in the following section.

A.4 Hierarchical grids

As was noted in the previous section, the size of an object relative to each voxel in a grid influences how many voxels will contain that object. This in turn negatively affects insertion and deletion times. Hence, it would make sense to find a spatial subdivision whereby the voxels can have different sizes. If this is accomplished, then insertion and deletion of objects can be made independent of their sizes and can therefore be executed in constant time. Such spatial subdivisions are not new and are known as hierarchical spatial subdivisions. Octrees, bintrees and hierarchical grids are all examples of hierarchical spatial subdivisions. However, normally such spatial subdivisions store

all their objects in leaf nodes and would therefore still incur non-constant insertion and deletion costs. We extend the use of hierarchical grids in such a way that objects can also reside in intermediary nodes or even in the root node for objects that are nearly as big as the entire scene.

Because such a structure should also be able to deal with expanding scenes, our efforts were directed towards constructing a hierarchy of grids (similar to Sung [24]), thereby extending the functionality of the grid structure presented in the previous section. Effectively, the proposed method constitutes a balanced octree.

Object insertion now proceeds similarly to grid insertion, except that the grid level needs to be determined before insertion. This is accomplished by comparing the size of the object in relation to the size of the scene. A simple heuristic is to determine the grid level from the diagonals of the two bounding boxes. Specifically, the length of the grid's diagonal is divided by the length of the object's diagonal, the result determining the grid level. Insertion and deletion progresses as explained in the previous section.

The gain of constant time insertion is offset by a slightly more complicated traversal algorithm. Hierarchical grid traversal is effectively the same as grid traversal with the following modifications. Traversal always starts at a leaf node which may first be mapped to a physical leaf node as described in the previous section. The ray is intersected with this voxel and all its parents until the root node is reached. This is necessary because objects at all levels in the hierarchy may occupy the same space as the currently traversed leaf node. If an intersection is found within the space of the leaf node, then traversal is finished. If not, the next leaf node is selected and the process is repeated.

This traversal scheme is wasteful because the same parent nodes may be repeatedly traversed for the same ray. To combat this problem, note that common ancestors of the current leaf node and the previously intersected leaf node, need not be traversed again. If the ray direction is positive, the current voxel's number can be used to derive the number of levels to go up in the tree to find the common ancestor between the current and the previously visited voxel. For negative ray directions, the previously visited voxel's number is used instead. Finding the common ancestor is achieved using simple bit manipulation, as detailed in Figure A.2.

```
bitmask = (raydir_x > 0) ? x : x + 1
forall levels in hierarchical grid
{
  cell = hgrid[level][x>>level][y>>level][z>>level]
  forall objects in cell
    intersect(ray, object)
  if (bitmask & 1)
    return
  bitmask >>= 1
}
```

Figure A.2: Hierarchical grid traversal algorithm in C-like pseudo-code. The bitmask is set assuming that the last step was along the x-axis.

As the highest levels of the grid may not contain any objects, ascending all the way to the highest level in the grid is not always necessary. Ascending the tree for a

particular leaf node can stop when the largest voxel containing objects is visited.

This hierarchical grid structure has the following features. The traversal is only marginally more complex than standard grid traversal. In addition, wrapping of objects in the face of expanding scenes is still possible. If all objects are the same size, this algorithm effectively defaults to grid traversal. Insertion and deletion can be achieved in constant time, as the number of voxels that each object overlaps is roughly constant[1].

A.5 Evaluation

The grid and hierarchical grid spatial subdivisions were implemented using an interactive ray tracer [17], which runs on an SGI Origin 2000 with 32 processors. For evaluation purposes, two test scenes were used. In each scene, a number of objects were animated using pre-programmed motion paths. The scenes as they are at start-up are depicted in Figure A.5 (top). An example frame taken during the animation is given for each scene in Figure A.5 (bottom). All images were rendered on 30 processors at a resolution of 512^2 pixels.

To assess basic traversal speed, the new grid and hierarchical grid implementations are compared with a bounding volume hierarchy. We also compared our algorithms with a grid traversal algorithm which does not allow interactive updates. Its internal data structure consists of a single array of object pointers, which improves cache efficiency on the Origin 2000.

From here on we will refer to the new grid implementation as 'interactive grid' to distinguish between the two grid traversal algorithms. As all these spatial subdivision methods have a user defined parameter to set the resolution (voxels along one axis and maximum number of grid levels, respectively), various settings are evaluated. The overall performance is given in Figure A.3 and is measured in frames per second.

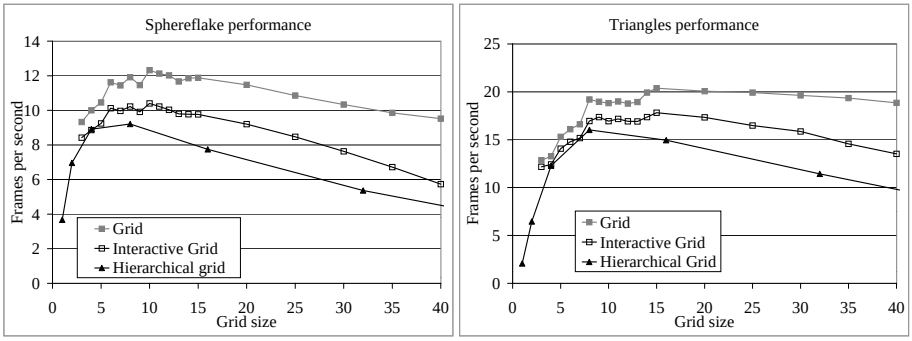


Figure A.3: Performance (in frames per second) for the grid, the interactive grid and the hierarchical grid for two static scenes. The bounding volume hierarchy achieves a frame rate of 8.5 fps for the static sphereflake model and 16.4 fps for the static triangles model.

[1]Note that this also obviates the need for mailbox systems to avoid redundant intersection tests.

The extra flexibility gained by both the interactive grid and hierarchical grid implementations results in a somewhat slower frame rate. This is according to expectation, as the traversal algorithm is a little more complex and the Origin's cache structure cannot be exploited as well with either of the new grid structures. The graphs in Figure A.3 show that with respect to the grid implementation the efficiency reduction is between 12% and 16% for the interactive grid and 21% and 25% for the hierarchical grid. These performance losses are deemed acceptable since they result in far better overall execution than dynamically reconstructing the original grid. For the sphereflake, all implementations are faster, for a range of grid sizes, than a bounding volume hierarchy, which runs at 8.5 fps. For the triangles scene, the hierarchical grid performs at 16.0 fps similarly to the bounding volume (16.4 fps), while grid and interactive grid are faster.

The non-zero cost of updating the scene effectively limits the number of objects that can be animated within the time-span of a single frame. However, for both scenes, this limit was not reached. In the case where the frame rate was highest for the triangles scene, updating all 200 triangles took less than 1/680th of a frame for the hierarchical grid and 1/323th of a frame for the interactive grid. The sphereflake scene costs even less to update, as fewer objects are animated. For each of these tests, the hierarchical grid is more efficiently updated than the interactive grid, which confirms its usefulness.

The size difference between different objects should cause the update efficiency to be variable for the interactive grid, while remaining relatively constant for the hierarchical grid. In order to demonstrate this effect, both the ground plane and one of the triangles in the triangle scene was interactively repositioned during rendering. The update rates for different size parameters for both the interactive grid and the hierarchical grid, are presented in Figure A.4 (left). As expected, the performance of the hierarchical grid is relatively constant, although the size difference between ground plane and triangle is considerable. The interactive grid does not cope with large objects very well if these objects overlap with many voxels. Dependent on the number of voxels in the grid, there is one to two orders of magnitude difference between inserting a large and a small object. For larger grid sizes, the update time for the ground plane is roughly half a frame. This leads to visible artifacts when using an interactive grid, as during the update the processors that are rendering the next frame temporarily cannot intersect this object (it is simply taken out of the spatial subdivision). In practice, the hierarchical grid implementation does not show this disadvantage.

The time to rebuild a spatial subdivision from scratch is expected to be considerably higher than the cost of re-inserting a small number of objects. For the triangles scene, where 200 out of 201 objects were animated, the update rate was still a factor of two faster than the cost of completely rebuilding the spatial subdivision. This was true for both the interactive grid and the hierarchical grid. A factor of two was also found for the animation of 81 spheres in the sphereflake scene. When animating only 9 objects in this scene, the difference was a factor of 10 in favor of updating. We believe that the performance difference between rebuilding the acceleration structure and updating all objects is largely due to the cost of memory allocation, which occurs when rebuilding.

In addition to experiments involving grids and hierarchical grids with a branching factor of two, tests were performed using a hierarchical grid with a higher branching factor. Instead of subdividing a voxel into eight children, here nodes are split into 64

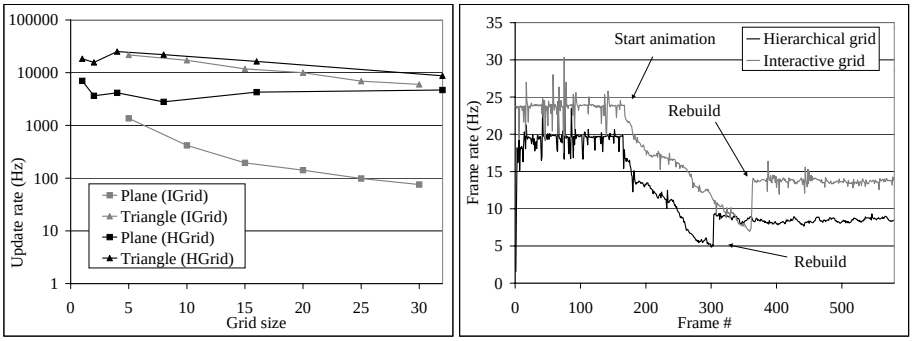


Figure A.4: Left: Update rate as function of (hierarchical) grid size. The plane is the ground plane in the triangles scene and the triangle is one of the triangles in the same scene. Right: Frame rate as function of time for the expanding triangle scene.

children (4 along each axis). The observed frame rates are very similar to the hierarchical grid. The object update rates were slightly better for the sphereflake and triangle scenes, because the size differences between the objects matches this acceleration structure better than both the interactive grid and the hierarchical grid.

In the case of expanding scenes, the logical bounding box will become larger than the physical bounding box. The number of voxels that are traversed per ray will therefore on average increase. This is the case in the triangles scene[2]. The variation over time of the frame rate is given in Figure A.4 (right). In this example, the objects are first stationary. At some point the animation starts and the frame rate drops because the scene immediately starts expanding. At some point the expansion is such that a rebuild is warranted. The re-computed spatial subdivision now has a logical bounding box which is identical to the (new) physical bounding box and therefore the number of traversed voxels is reduced when compared with the situation just before the rebuild. The total frame rate does not reach the frame rate at the start of the computation, because the objects are more spread out over space, resulting in larger voxels and more intersection tests which do not yield an intersection point.

Finally, Figure A.6 shows that interactively updating scenes using drag and drop interaction is feasible.

A.6 Conclusions

When objects are interactively manipulated and animated within a ray tracing application, much of the work that is traditionally performed during a pre-processing step becomes a limiting factor. Especially spatial subdivisions which are normally built once before the computation starts, do not exhibit the flexibility that is required for animation. The insertion and deletion costs can be both unpredictable and variable. We

[2]For this experiment, the ground plane of the triangles scene was reduced in size, allowing the rebuild to occur after a smaller number of frames.

have argued that for a small cost in traversal performance flexibility can be obtained and insertion and deletion of objects can be performed in constant time.

By logically extending the (hierarchical) grids into space, these spatial subdivisions deal with expanding scenes rather naturally. For modest expansions, this does not significantly alter the frame rate. When the scenes expand a great deal, rebuilding the entire spatial subdivision may become necessary. For large scenes this may involve a temporary drop in frame rate. For applications where this is unacceptable, it would be advisable to perform the rebuilding within a separate thread (rather than the display thread) and use double buffering to minimize the impact on the rendering threads.

Acknowledgements

Thanks to Pete Shirley and Steve Parker for their help and comments and to the anonymous reviewers for their helpful comments. This work was supported by NSF grants CISE-CCR 97-20192, NSF-9977218 and NSF-9978099 and by the DOE Advanced Visualization Technology Center.

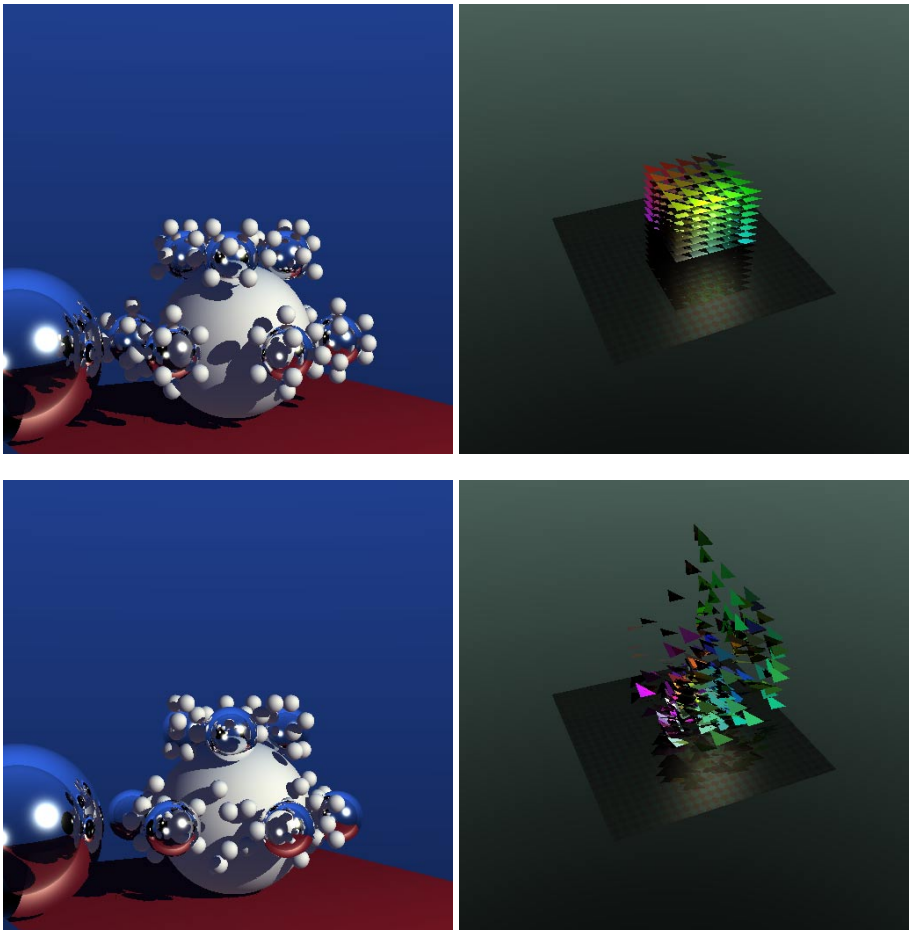


Figure A.5: Test scenes before any objects moved (top) and during animation (bottom).

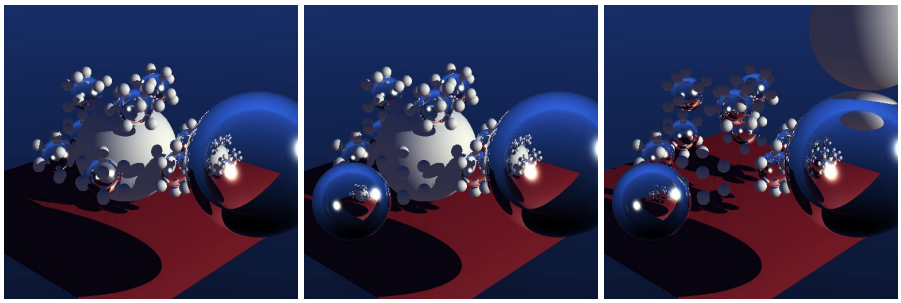


Figure A.6: Frames created during interactive manipulation.

Bibliography

[1] J. AMANATIDES AND A. WOO, *A fast voxel traversal algorithm for ray tracing*, in Eurographics '87, Elsevier Science Publishers, Amsterdam, North-Holland, Aug. 1987, pp. 3–10.

[2] K. BALA, J. DORSEY, AND S. TELLER, *Interactive ray traced scene editing using ray segment tree*, in Rendering Techniques '99, D. Lischinski and G. W. Larson, eds., Eurographics, Springer-Verlag Wien New York, 1999, pp. 31–44.

[3] ——, *Radiance interpolants for accelerated bounded-error ray tracing*, ACM Transactions on Graphics, (1999).

[4] G. BISHOP, H. FUCHS, L. MCMILLAN, AND E. J. SCHER ZAGIER, *Frameless rendering: Double buffering considered harmful*, in Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994), A. Glassner, ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, July 1994, pp. 175–176. ISBN 0-89791-667-0.

[5] J. G. CLEARY AND G. WYVILL, *Analysis of an algorithm for fast ray tracing using uniform space subdivision*, The Visual Computer, (1988), pp. 65–83.

[6] R. A. CROSS, *Interactive realism for visualization using ray tracing*, in Proceedings Visualization '95, 1995, pp. 19–25.

[7] A. FUJIMOTO, T. TANAKA, AND K. IWATA, *ARTS: Accelerated ray tracing system*, IEEE Computer Graphics and Applications, 6 (1986), pp. 16–26.

[8] V. GAEDE AND O. GÜNTHER, *Multidimensional access methods*, ACM Computing Surveys, 30 (1998), pp. 170–231.

[9] A. S. GLASSNER, *Space subdivision for fast ray tracing*, IEEE Computer Graphics and Applications, 4 (1984), pp. 15–22.

[10] A. S. GLASSNER, *Spacetime ray tracing for animation*, IEEE Computer Graphics and Applications, 8 (1988), pp. 60–70.

[11] A. S. GLASSNER, ed., *An Introduction to Ray Tracing*, Academic Press, San Diego, 1989.

[12] G. W. LARSON AND M. SIMMONS, *The holodeck ray cache: An interactive rendering system for global illumination in non-diffuse environments*, ACM Transactions on Graphics, 18 (October 1999), pp. 361–368.

[13] J. D. MACDONALD AND K. S. BOOTH, *Heuristics for ray tracing using space subdivision*, The Visual Computer, (1990), pp. 153–166.

[14] M. J. MUUSS, towards real-time ray-tracing of combinatorial solid geometric models, in Proceedings of BRL-CAD Symposium, June 1995.

[15] K. NAKAMARU AND Y. OHNO, *Breadth-first ray tracing utilizing uniform spatial subdivision*, IEEE Transactions on Visualization and Computer Graphics, 3 (1997), pp. 316–328.

[16] K. NEMOTO AND T. OMACHI, *An adaptive subdivision by sliding boundary surfaces for fast ray tracing*, in Proceedings of Graphics Interface '86, M. Green, ed., May 1986, pp. 43–48.

[17] S. PARKER, W. MARTIN, P.-P. SLOAN, P. SHIRLEY, B. SMITS, AND C. HANSEN, *Interactive ray tracing*, in Symposium on Interactive 3D Computer Graphics, April 1999.

[18] S. PARKER, M. PARKER, Y. LIVNAT, P.-P. SLOAN, C. HANSEN, AND P. SHIRLEY, *Interactive ray tracing for volume visualization*, in IEEE Transactions on Visualization and Computer Graphics, July-September 1999.

[19] E. REINHARD, P. SHIRLEY, AND C. HANSEN, *Parallel point reprojection*. Submitted to IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics.

[20] E. REINHARD, B. SMITS, AND C. HANSEN, *Dynamic acceleration structures for interactive ray tracing*, in Proceedings of the 11th Eurographics Workshop on Rendering, Brno, Czech Republic, June 2000, pp. 299–306.

[21] M. SIMMONS AND C. SÉQUIN, *Tapestry: A dynamic mesh-based display representation for interactive rendering*, in Proceedings of the 11th Eurographics Workshop on Rendering, Brno, Czech Republic, June 2000, pp. 329–340.

[22] B. SMITS, *Efficiency issues for ray tracing*, Journal of Graphics Tools, 3 (1998), pp. 1–14.

[23] J. SPACKMAN AND P. WILLIS, *The SMART navigation of a ray through an octtree*, Computers and Graphics, 15 (1991), pp. 185–194.

[24] K. SUNG, *A DDA octree traversal algorithm for ray tracing*, in Eurographics '91, W. Purgathofer, ed., North-Holland, sept 1991, pp. 73–85. European Computer Graphics Conference and Exhibition; held in Vienna, Austria; 2-6 September 1991.

[25] B. WALTER, G. DRETTAKIS, AND S. PARKER, *Interactive rendering using the render cache*, in Rendering Techniques '99, D. Lischinski and G. W. Larson, eds., Eurographics, Springer-Verlag Wien New York, 1999, pp. 19–30.

[26] K.-Y. WHANG, J.-W. SONG, J.-W. CHANG, J.-Y. KIM, W.-S. CHO, C.-M. PARK, AND I.-Y. SONG, *Octree-r: An adaptive octree for efficient ray tracing*, IEEE Transactions on Visualization and Computer Graphics, 1 (1995), pp. 343–349.

[27] E. S. ZAGIER, *Defining and refining frameless rendering*, Tech. Rep. TR97-008, UNC-CS, July 1997.